■ ไพโรจน์ ไววานิชกิจ

Shell Programming กับ UNIX ตอนที่ 4

แนะนำคำสั่งควบคุมเงื่อนไขที่มีอยู่ในระบบปฏิบัติการ UNIX บนเชลล์แบบบอร์น ต่อจากตอนที่แล้ว

สำหรับบทความในตอนที่ 4 ผู้เขียนจะขอแนะนำคำสั่งควบคุมเงื่อนไขที่มีอยู่ในระบบปฏิบัติการ UNIX บนเชลล์แบบบอร์น (Bourne Shell) ต่อจากบทความในตอนที่แล้ว อนึ่ง ผู้อ่านที่ใช้งานระบบปฏิบัติการ LINUX สามารถทดลองฝึกหัดเขียนเชลล์สคริปต์ได้โดยการกำหนดเชลล์ใช้งานของตนเป็นเชลล์แบบบอร์น ในกรณีที่ใช้งานอยู่บนเชลล์แบบซี ก็สามารถปรับเชลล์ของตนเป็นแบบบอร์นได้ด้วยการพิมพ์ sh

คำสั่ง for In และ: for

คำสั่งควบคุมในตระกูล for บนระบบปฏิบัติการ UNIX มีรูปแบบการใช้งานในลักษณะเดียวกับคำสั่ง for () ในภาษาซีหรือคำสั่ง for to ในภาษาปาสคาล ในการใช้งาน ผู้ใช้งานจะทำการกำหนดขอบเขตของตัวแปรหรือเงื่อนไขต่าง ๆ ที่แน่ชัดขึ้นเพื่อสร้างวงรอบการทำงาน (Loop) ให้กับโปรแกรม สำหรับรูปแบบโครงสร้างของคำสั่ง for in แสดงได้ดังนี้

```
for loop-index in argument-list
do
    commands
done
```

โดยเมื่อมีการเรียกใช้งานรูทีนดังกล่าว เชลล์จะทำการดึงค่าข้อมูลแต่ละตัวที่ได้รับการระบุไว้ใน argument-list มาแทนค่าลงในตัวแปร loop-index เริ่มตั้งแต่ค่าแรกไปจนถึงค่าสุดท้าย โดยเมื่อมีการแทนค่าตัวแปรตัวแรกก็จะมีการปฏิบัติงานตามคำสั่งซึ่งได้รับการระบุไว้ระหว่างสเตทเมนต์ do และ done เมื่อปฏิบัติคำสั่งไปจนถึงสเตทเมนต์ done แล้ว เชลล์จะทำการกำหนดค่าถัดไปใน argument -list ให้กับตัวแปร loop-index เพื่อดำเนินการวนรอบใหม่ต่อไปเรื่อย ๆ จนกระทั่งถึงค่าสุดท้าย

สำหรับตัวอย่างในกรอบที่ 1 เป็นการทดลองใช้งานคำสั่งควบคุม for in โดยกำหนดให้ argument -list ประกอบด้วยสมาชิกจำนวน 4 ชุด ด้วยกัน

UNIX

กรอบที่ 1

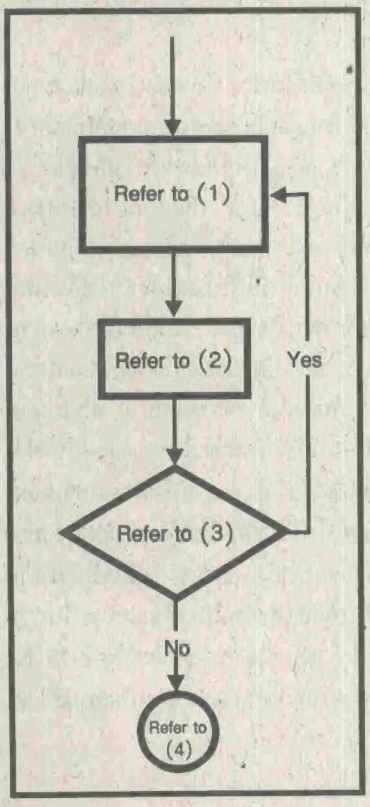
```
$ cat province
for province in Chiangmai Bangkok Hatyai Khonkaen
do
    echo $province
done
echo These are name of major province in Thailand.

$ province
Chiangmai
Bangkok
Hatyai
Khonkaen
These are name of major province in Thailand.
```

ในกรณีที่ผู้ใช้งานต้องการป้อนอาร์กิวเมนต์ที่จะนำมาใช้เป็นขอบเขตเงื่อนไขได้ด้วยตนเองแทนที่การกำหนดค่าตายตัวดังตัวอย่างในกรอบที่ 1 ก็สามารถเลือกใช้คำสั่ง for ซึ่งสามารถทำงานได้โดยรับค่าอาร์กิวเมนต์จากการป้อนเข้าโดยตรง (Command line) ผ่านทางอุปกรณ์อินพุตมาตรฐานต่าง ๆ เช่น จากแป้นพิมพ์ จากการไปป์ หรือการเปลี่ยนทิศทาง เป็นต้น ในกรณีนี้เชลล์จะทำการอ่านตัวแปรพิเศษ $@ ซึ่งเก็บค่าอาร์กิวเมนต์ต่าง ๆ ที่ผู้ใช้งานป้อนผ่านเข้ามา ขอให้ลองศึกษาตัวอย่างในกรอบที่ 2 ประกอบ สำหรับรูปแบบโครงสร้างของคำสั่ง for เป็นดังนี้

```
for loop-index
do
    commands
done
```

รูปที่ 1 แผนภาพการทำงานของคำสั่งควบคุม for in

(1) กำหนดค่าอาร์กิวเมนต์ตัวถัดไปใน argument-list ให้กับตัวแปร loop-index
(2) ดำเนินการตามคำสั่ง commands
(3) ตรวจสอบว่าค่าอาร์กิวเมนต์ที่ใช้อยู่เป็นค่าสุดท้ายใน argument-list หรือไม่
(4) สิ้นสุดการดำเนินการ



กรอบที่ 2

```
$ cat for_test
for args
do
    echo $args
done
$ for_test Chiangmai Bangkok Hatyai Khonkaen
Chiangmai
Bangkok
Hatyai
Khonkaen
```

คำสั่ง while

เป็นอีกคำสั่งหนึ่งที่ใช้ในการควบคุมวงรอบการทำงานโดยเชลล์จะทำการปฏิบัติตามคำสั่งภายในวงรอบการทำงานไปเรื่อย ๆ จนกว่านิพจน์ที่กำหนดไว้ให้กับคำสั่ง while จะแสดงค่าความจริงออกมาเป็นเท็จ ตราบใดที่ค่านิพจน์ที่ตามหลังคำสั่ง while ยังให้ค่าความจริงเป็นจริง (true) อยู่ การวนรอบทำงานก็ยังคงเกิดขึ้นต่อไป รูปแบบการใช้งานคำสั่ง while เป็นดังนี้

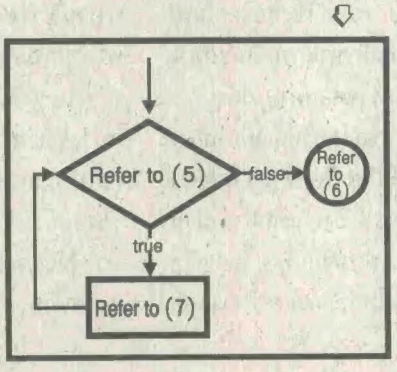
```
while test-command
do
    commands
done
```

นิพจน์ที่ผู้เขียนกล่าวถึงก็คือ test-command ซึ่งอาจกำหนดให้เป็นตัวแปรอย่างง่ายไปจนถึงรูปแบบนิพจน์ที่มีความซับซ้อน ทั้งนี้ขึ้นอยู่กับการออกแบบและความจำเป็นในการออกแบบโปรแกรมของผู้ใช้งานเอง ตราบใดที่นิพจน์ test-command ยังมีค่าความจริงเป็น true อยู่ เชลล์ก็จะคงวนรอบดำเนินการปฏิบัติตามคำสั่งที่อยู่ระหว่างสเตทเมนต์ do และ done ต่อไป สำหรับรูปแบบการจัดโครงสร้างของคำสั่ง while แสดงในรูปที่ 2

เชลล์สคริปต์ในกรอบที่ 3 เป็นการกำหนดให้มีการแสดงผลค่าตัวเลขตั้งแต่ 0 ถึง 9 บนจอภาพโดยใช้คำสั่ง while ขอให้ผู้อ่านทุกท่านทดลองเขียนสคริปต์ดังกล่าวโดยใช้คำสั่งควบคุม for in แทน while ดูครับ สำหรับการทำงานของสคริปต์จะเริ่มจากกำหนดค่าเริ่มต้นให้ตัวแปร number มีค่าเป็น 0 จากนั้นจึงเริ่มเข้าสู่วงรอบการทำงานของคำสั่ง while ในกรณีนี้นิพจน์ที่ใช้ในการตรวจสอบจะใช้คำสั่ง test ซึ่งแทนด้วยเครื่องหมาย [ ] ซึ่งทำหน้าที่ตรวจสอบว่าค่าของตัวแปร number มีค่าน้อยกว่า 10 หรือไม่ อาร์กิวเมนต์ -lt ที่เห็นเป็นการกำหนดประเภทของการทดสอบ โดยค่าที่เป็นได้มีตั้งแต่ -ne (not equal ไม่เท่ากับ) -eq (equal เท่ากับ) -gt (greater than มากกว่า) -ge (greater than or

รูปที่ 2 แผนภาพการทำงานของคำสั่งควบคุม while

(5) ค่าความจริงของนิพจน์ test-command เป็นอย่างไร
(6) สิ้นสุดการดำเนินการ
(7) ดำเนินการตามคำสั่ง commands



UNIX

equal เท่ากับหรือมากกว่า) –lt (less than น้อยกว่า) และ –le (less than or equal น้อยกว่า หรือเท่ากับ) ทั้งหมดนี้ใช้สำหรับการทดสอบตัวแปรประเภทตัวเลข สำหรับการตรวจสอบตัวแปรที่เป็นกลุ่มตัวอักษรก็จะสามารถระบุเป็น = (equal เหมือนกัน) หรือ != (not equal ต่างกัน) เมื่อกำหนดเงื่อนไขในการตรวจสอบแล้ว คำสั่ง test จะส่งค่ากลับคืนมาเป็น true เมื่อเงื่อนไขในการตรวจสอบยังเป็นจริงอยู่ สำหรับส่วนที่เหลือก็คงจะสามารถทำความเข้าใจได้ไม่ยากนัก

กรอบที่ 3

```
$ cat count
number=0
while [ "$number" - lt 10]
do
    echo "$number"
    number='expr $number + 1'
done
echo

$ count
0123456789
$
```

ลองมาดูตัวอย่างการประยุกต์ใช้งานที่ซับซ้อนมากขึ้นในกรอบที่ 4 เป็นเชลล์สคริปต์ชื่อ spell_check ซึ่งใช้ตรวจสอบคำสะกดในไฟล์ข้อความ (text file) ใด ๆ ตามที่ต้องการ อันที่จริงระบบปฏิบัติการ UNIX เองก็มียูทิลิตี้ชื่อ spell ซึ่งทำหน้าที่ตรวจสอบคำสะกดโดยเทียบคำศัพท์ตามพจนานุกรมมาตรฐานอยู่แล้ว แต่ในกรณีของคำเฉพาะ เช่น ชื่อบริษัท ชื่อสถานที่ หรือศัพท์เฉพาะในแต่ละวงการ ผู้ใช้งานมักไม่ต้องการให้คำสั่ง spell แสดงความผิดพลาดขึ้น สคริปต์ spell check มีหน้าที่ตรวจสอบคำสะกดโดยใช้ความสามารถของคำสั่ง spell สำหรับในกรณีของคำเฉพาะนั้นจะมีการกำหนดให้ผู้ใช้งานสร้างไฟล์ข้อความซึ่งบรรจุคำเฉพาะไว้และให้ป้อนเป็นอินพุตตัวหนึ่งในการเรียกใช้คำสั่งด้วย ในที่นี้จะขอเรียกชื่อไฟล์ดังกล่าวว่าไฟล์พจนานุกรมศัพท์เฉพาะ ทั้งนี้เชลล์สคริปต์จะแสดงรายการคำที่สะกดผิดพลาดซึ่งคำสั่ง spell ตรวจไม่พบและไม่ปรากฏในไฟล์พจนานุกรมศัพท์เฉพาะ สำหรับการเรียกใช้งานจะอยู่ในรูป spell_check ตามด้วยชื่อของไฟล์พจนานุกรมศัพท์เฉพาะและชื่อไฟล์ข้อความที่ต้องการตรวจสอบตามลำดับ

จากกรอบที่ 4 จะเห็นว่าเชลล์สคริปต์เริ่มจากการตรวจสอบจำนวนอาร์กิวเมนต์ที่ป้อนผ่านเข้ามา โดยการใช้คำสั่ง test ร่วมกับตัวแปร $# ซึ่งจะแทนค่าจำนวนอาร์กิวเมนต์ที่ป้อนผ่านมาทางอุปกรณ์อินพุต ในกรณีที่ผู้ใช้งานป้อนชื่อไฟล์ครบทั้งสองไฟล์ ก็จะมีการตรวจสอบต่อไปว่าเชลล์สามารถเปิดอ่านแต่ละไฟล์ได้หรือไม่ ทั้งนี้ด้วยการใช้คำสั่ง test ตามด้วยอาร์กิวเมนต์ –r (readable) ซึ่งจะส่งค่ากลับมาเป็น true ถ้าเปิดอ่านไฟล์ที่ถูกระบุชื่อไว้ได้ เมื่อผ่านการตรวจสอบทั้ง 2 ขั้นตอนแล้ว ก็จะมีการเรียกใช้คำสั่ง spell เพื่อแสดงรายการคำศัพท์ที่ไม่พบในพจนานุกรมมาตรฐาน (ติดตั้งมาพร้อมกับระบบปฏิบัติการ UNIX) รายการคำศัพท์ทั้งหมดจะถูกส่งต่อด้วยการไปป์ไปยังคำสั่ง while

ก่อนจะอธิบายต่อไป ผู้เขียนขอกล่าวถึงรูปแบบผลลัพธ์ที่ได้จากคำสั่ง spell เสียก่อน โดยทั่วไปเมื่อมีการเรียกใช้คำสั่ง spell และมีการตรวจพบคำที่ไม่ปรากฏในพจนานุกรมมาตรฐาน คำสั่ง spell จะส่งรายการคำเหล่านั้นมาปรากฏบนจอภาพหรืออุปกรณ์มาตรฐานอื่น ๆ แล้วแต่กำหนด โดยเรียงบรรทัดละคำ อันมีตัวอย่างดังนี้

```
satruday
mafazine
recodder
```

จะเห็นได้ว่าผลลัพธ์ที่ได้มีการจัดเรียงคำศัพท์ออกมาเป็นบรรทัด เนื่องจากอักขระที่ใช้ในการเริ่มต้นและสิ้นสุดบรรทัดแต่ละบรรทัดคือ ^ และ $ ตามลำดับ ประกอบกับอักขระที่ใช้แจ้งการสิ้นสุดบรรทัดสุดท้ายคือ EOF (เราจะไม่เห็นอักขระควบคุมเหล่านี้บนจอภาพ) การจัดนิพจน์สำหรับคำสั่ง while จึงอยู่ในรูปของ while read line ซึ่ง line เป็นตัวแปรที่รับค่าข้อความซึ่งเป็นผลลัพธ์ของคำสั่ง spell

สำหรับคำศัพท์ที่ผ่านเข้ามาในวงรอบ while จะถูกนำมาเทียบค่าว่ามีปรากฏอยู่ในไฟล์พจนานุกรมศัพท์เฉพาะหรือไม่โดยใช้คำสั่ง grep ให้สังเกตการอ้างอิงคำศัพท์แต่ละบรรทัดซึ่งอยู่ในรูป "^$line" เพื่อเป็นการสร้างความมั่นใจว่าได้ดึงตัวอักษรมาครบทุกคำในบรรทัดนั้นแล้ว หากคำสั่ง grep พบว่าคำที่ได้รับมามีปรากฏอยู่ในไฟล์พจนานุกรมศัพท์เฉพาะก็จะทำการเปลี่ยนทิศทางการแสดงผลของคำ ๆ นั้นไปยัง null หรือก็คือไม่แสดงผลนั่นเอง พร้อม ๆ กับแสดงค่าแจ้งกลับเป็น 0 ซึ่งหมายถึงทำงานสำเร็จ ในกรณีที่คำดังกล่าวไม่ปรากฏอยู่ในไฟล์พจนานุกรมศัพท์เฉพาะ คำสั่ง grep จะให้ค่าแจ้งกลับเป็น 1 ซึ่งในกรณีนี้จะทำให้มีการพิมพ์คำดังกล่าวออกมาบนหน้าจอ การทำงานของคำสั่ง while จะดำเนินเป็นวงรอบต่อไป จนกระทั่งเสร็จสิ้นคำศัพท์บรรทัดสุดท้าย ในการวนรอบครั้งต่อไป ค่าของตัวแปร line จะเป็น EOF ซึ่งจะทำให้นิพจน์มีค่าความจริงเป็น false และโปรแกรมสิ้นสุดการทำงาน

UNIX

จากตัวอย่าง ผู้เขียนทดลองสร้างไฟล์พจนานุกรมมาตรฐานมีชื่อว่า word_list เพื่อใช้เป็นไฟล์อ้างอิงสำหรับการตรวจสอบคำสะกด การทดสอบเชลล์สคริปต์ spell_check จะกระทำโดยป้อนไฟล์ข้อความชื่อ spell_memo ซึ่งมีคำที่ไม่พบในพจนานุกรมมาตรฐานอยู่ 4 คำ มี 2 คำเป็นคำเฉพาะซึ่งมีระบุอยู่ในไฟล์ word_list ผลที่ได้จากการทดลองใช้เชลล์สคริปต์ spell_check จึงแสดงรายการคำที่สะกดผิดออกมาเพียง 2 คำที่เหลือเท่านั้น

กรอบที่ 4

```
$ cat spell_check
# remove correct spellings from spell out

if [ $# != 2 ]
 then
     echo "Usage: spell_check file1 file2" 1>&2
     echo "file1: list of correct spellings" 1>&2
     echo "file2: file to be checked" 1>&2
fi

if [ ! -r "$1" ]
 then
     echo "spell_check: $1 is not readable" 1>&2
     exit 1
fi

if [ ! -r "$2" ]
 then
     echo "spell_check: $2 is not readable" 1>&2
fi

spell "$2" |
while read line
do
 grep "^$line" "$1" > /dev/null
 if [ $? != 0 ]
     then
      echo $line
     fi
done

$ spell memo
Se-Education
Asoke-Dindaeng
targat
hte
$ cat word_list
Se-Education
Asoke-Dindaeng
$ spell_check word_list memo
targat
hte
```

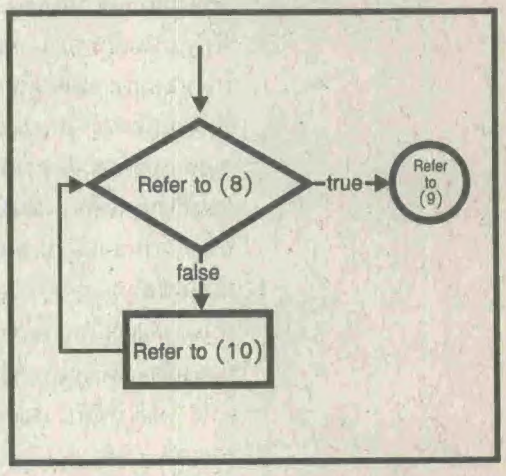
คำสั่ง until

คำสั่งควบคุม until มีรูปแบบโครงสร้างการใช้งานคล้ายคลึงกับคำสั่งควบคุม while ค่อนข้างมาก ความแตกต่างประการเดียวก็คือลำดับขั้นตอนการเข้าสู่วงรอบการทำงาน จากรูปที่ 3 อันแสดงแผนภาพการทำงานของคำสั่ง unitl จะเห็นว่าเมื่อนิพจน์ในการตรวจสอบให้ค่าความจริงเป็น true ก็จะทำให้เชลล์ออกจากการปฏิบัติงานตามวงรอบไปในทันที ต่างจากคำสั่ง while ที่จะต้องปฏิบัติงานตามคำสั่งในวงรอบเสียก่อน ก่อนที่จะหลุดออกไป การจัดรูปแบบการใช้งานของคำสั่ง until เป็นดังนี้

```
until test-command
do
  commands
done
```

ตัวอย่างในกรอบที่ 5 แสดงการใช้งานคำสั่ง until ร่วมกับคำสั่ง read โดยนิพจน์ที่ใช้กำหนดการวนรอบจะเป็นการเปรียบเทียบระหว่างค่าที่ผู้ใช้งานป้อนผ่านเข้ามาทางอุปกรณ์อินพุตมาตรฐานซึ่งในที่นี้ก็คือคีย์บอร์ดกับค่ารหัส (secretname) ที่ได้กำหนดไว้ในโปรแกรมแล้วเป็นสคริปต์ตัวอย่างง่าย ๆ สำหรับตัวอย่างในกรอบที่ 6 เป็นสคริปต์ของโปรแกรมชื่อ locktty ซึ่งใช้ในการป้องกันการลักลอบสวมรอยใช้อุปกรณ์เทอร์มินัลในขณะที่ผู้ใช้งานไม่อยู่ที่เครื่องของตนชั่วคราว

รูปที่ 3 แผนภาพการทำงานของคำสั่งควบคุม until
(8) ค่าความจริงของนิพจน์ test-command เป็นอย่างไร
(9) สิ้นสุดการดำเนินการ
(10) ดำเนินการตามคำสั่ง commands



UNIX

กรอบที่ 5
```
$ cat until1
secretname=sailboat
name=noname
echo Try to guess the secret name...
echo
until [ "$name" = "$secretname" ]
do
      echo "Your guess: "
      read name
done
echo Excellent, You hit it !

$ until1
Try to guess the secret name...
Your guess: sea
Your guess: pattaya
Your guess: sailboat
Excellent, You hit it !
```

กรอบที่ 6
```
$ cat locktty

trap '' 1 2 3 18
stty -echo
echo "Key: "
read key_1
echo
echo "Again: "
read key_2
echo
key_3=
if [ "$key_1" = "$key_2" ]
      then
        tput clear
        until [ "$key_3" = "$key_2" ]
        do
          read key_3
        done
      else
        echo "locktty: keys do not match" 1>&2
fi
stty echo
```

สำหรับตัวอย่างในกรอบที่ 6 เป็นเชลล์สคริปต์ซึ่งให้ผลการทำงานในลักษณะเดียวกับคำสั่ง lock บนระบบปฏิบัติ Berkeley UNIX โดยเมื่อผู้ใช้งานไม่อยู่ที่เครื่องของตนก็จะสั่งเรียกใช้สคริปต์คำสั่งดังกล่าว สคริปต์จะเริ่มทำงานโดยการสร้างวงรอบการทำงาน บุคคลอื่นที่ต้องการเข้าใช้เครื่องจะต้องทราบรหัสผ่าน ซึ่งผู้ใช้งานคนแรกเป็นผู้กำหนดในแต่ละครั้ง การทำงานเริ่มจากการใช้คำสั่ง trap เพื่อป้องกันการระงับการทำงานของโปรแกรมโดยการส่งสัญญาณสิ้นสุดการทำงาน (stop signal) ด้วยการกดปุ่มขัดจังหวะต่าง ๆ เช่น ปุ่ม delete หรือปุ่ม CONTROL-C จากผู้ใช้งานรายอื่น สำหรับสัญญาณหมายเลข 18 ที่มีการระบุไว้จะหมายถึงการป้องกันมิให้ผู้ใช้งานรายอื่นใช้คำสั่ง CONTROL-Z เพื่อหยุดยั้งการทำงานของโปรแกรม สำหรับคำสั่ง stty -echo จะทำให้เชลล์ไม่สะท้อนค่าตัวอักษรที่พิมพ์เข้าไปในขณะป้อนรหัสเพื่อสั่งให้โปรแกรมทำงาน อันเป็นการป้องกันผู้อื่นมองเห็นก่อนจะลุกออกไปจากเครื่องของตน ผู้ใช้งานจะเรียกใช้สคริปต์ดังกล่าวพร้อมกับป้อนรหัสที่ตนต้องการใช้ในการล็อกหน้าจอของตน เชลล์สคริปต์จะรับรหัสที่ป้อนในครั้งแรก พร้อมกับขอให้ป้อนใหม่อีกครั้งเพื่อเป็นการยืนยัน หากรหัสตรงกันก็จะลบภาพบนหน้าจอออก พร้อมกับเริ่มวนรอบ ผู้ที่จะเข้ามาใช้เครื่องต้องทำการป้อนรหัสปลดล็อก ซึ่งกระบวนการดังกล่าวอยู่ภายในวงรอบการทำงานของคำสั่ง until หากป้อนรหัสผ่านตรงกัน เชลล์ก็จะหลุดออกจากวงรอบ และปรับ echo ให้ทำการสะท้อนข้อความกลับบนหน้าจอตามเดิม

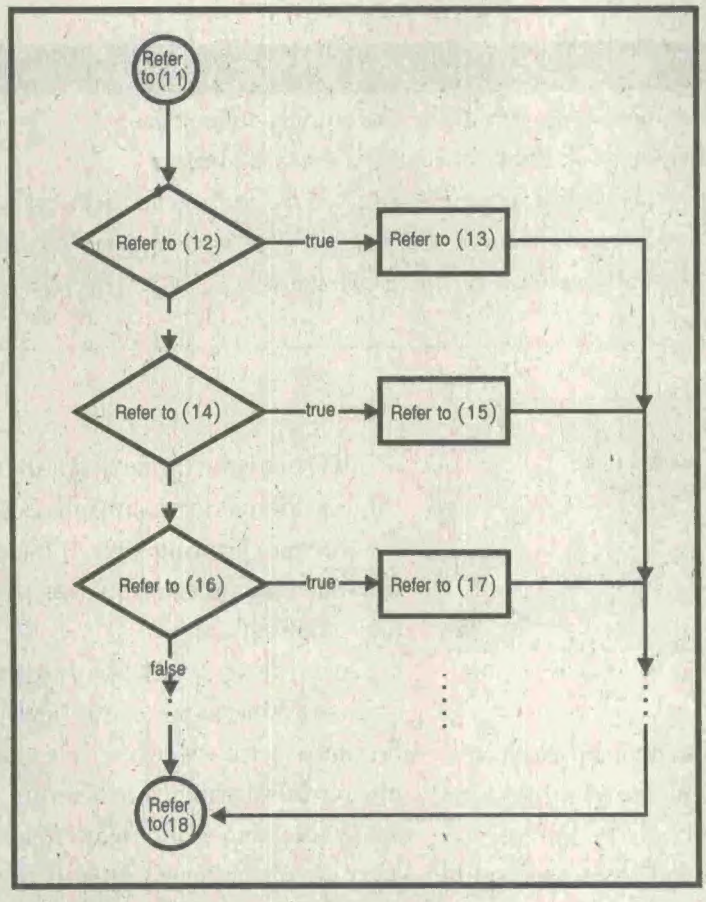
คำสั่ง case

คำสั่งควบคุมสุดท้ายที่จะกล่าวถึงก็คือคำสั่ง case ซึ่งใช้สำหรับการกำหนดทางเลือกที่มีมากกว่า 2 ทางขึ้นไปแทนที่จะใช้คำสั่ง if... then... สำหรับแผนภาพแสดงการทำงานของคำสั่งดังกล่าวแสดงอยู่ในรูปที่ 4 และการจัดโครงสร้างการใช้งานเป็นดังนี้

```
case test-string in
     pattern-1 )
       commands-1
       ;;
     pattern-2 )
       commands-2
       ;;
     pattern-3 )
       commands-3
       ;;

esac
```

จากรูปที่ 4 จะเห็นว่าการที่เชลล์จะเลือกดำเนินการปฏิบัติตามคำสั่งในทางเลือกใดนั้นก็ขึ้นอยู่กับค่าของตัวแปร test-string ว่าตรงกับเงื่อนไข pattern ค่าใด สำหรับตัวอย่างในกรอบที่ 7 จะแสดงการสร้างเงื่อนไขสำหรับปฏิบัติจำนวน 3 ทางเลือกโดยพิจารณาจากค่าของตัวแปร letter หากตัวแปรดังกล่าวมีค่า A ก็จะทำให้เชลล์ดำเนินการปฏิบัติการไปตามทางเลือก A และหากค่าตัวแปรตรงกับ B หรือ C ก็จะทำให้การปฏิบัติการเป็นไปตามทางเลือกที่เกี่ยวข้องกันไปตามแต่กรณี สำหรับเครื่องหมายดอกจัน (*) จะใช้สำหรับรองรับค่า

UNIX



รูปที่ 4 แผนภาพการทำงานของคำสั่งควบคุม case

(11) case
(12) ตรวจสอบค่าของ string ตรงกับ pattern-1 หรือไม่
(13) ดำเนินการตามคำสั่ง commands-1
(14) ตรวจสอบค่าของ string ตรงกับ pattern-1 หรือไม่
(15) ดำเนินการตามคำสั่ง commands-2
(16) ตรวจสอบค่าของ string ตรงกับ pattern-1 หรือไม่
(17) ดำเนินการตามคำสั่ง commands-3
(18) สิ้นสุดการดำเนินการ

อื่น ๆ ที่ถูกป้อนผ่านเข้ามาทางตัวแปร letter ซึ่งไม่สอดคล้องกับเงื่อนไขที่ได้กำหนดรองรับไว้ (ปกติจะเรียกว่าค่าโดยปริยาย หรือ Default value) ตารางที่ 1 แสดงเครื่องหมายต่าง ๆ ที่ผู้ใช้งานสามารถนำมาใช้ร่วมกับการอ้างอิงคำสั่ง case โดยมีคำอธิบายประกอบไว้อย่างชัดเจนแล้ว ขอให้ลองศึกษาดูครับ

สำหรับตัวอย่างสุดท้ายในกรอบที่ 8 เป็นเชลล์สคริปต์ชื่อ safedit ซึ่งทำงานโดยใช้ความสามารถของคำสั่ง case ในการตรวจสอบจำนวนอาร์กิวเมนต์ที่ผู้ใช้งานป้อนผ่านเข้ามา สคริปต์ดังกล่าวใช้งานในการสำรอง (backup) ไฟล์ข้อความที่ผู้ใช้งานสั่งทำการแก้ไข ในที่นี้จะเรียกใช้อีดิตเตอร์ vi

กรอบที่ 7

```
$ cat case_sample
echo "Enter A, B, or C: "
read letter
case "$letter" in
      A)
        echo You entered A
        ;;
      B)
        echo You entered B
        ;;
      C)
        echo You entered C
        ;;
      *)
        echo You did not enter A, B, or C
        ;;
esac

$ case_sample
  Enter A, B, or C: B
  You entered B
  $ case_sample
  Enter A, B, or C: b
  You did not enter A,B, or C
```

กรอบที่ 8

```
$ cat safedit
PATH=/usr/ucb:/bin:/usr/bin
script='basename $0'
case $# in
      0)
        vi
        exit 0
        ;;
      1)
        if [ ! -f "$1" ]
          then
            vi "$1"
            exit 0
        fi
        if [ ! -r "$1" -o ! -w "$1" ]
          then
            echo "$script: check permissions on $1" 1>&2
            exit 1
          else
            editfile=$1
        fi
        if [ ! -w "." ]
          then
            echo "$script: backup cannot be " \plain
"created in the working directory" 1>&2
            exit 1
        fi
        ;;

      *)
        echo "Usage: $script [file-to-edit]" 1>&2
        exit 1
        ;;
esac
tempfile=/tmp/$$.$script
cp $editfile $tempfile
if vi $editfile
```

UNIX

ตารางที่ 1 เครื่องหมายที่ใช้ร่วมกับคำสั่ง case

เครื่องหมาย	คำอธิบายและการใช้งาน
*	เครื่องหมายดอกจัน จะใช้แทนตัวกลุ่มคำหรือตัวเลขใด ๆ ทุกชนิด ใช้ในกรณีของค่าโดยปริยาย (Default value)
?	เครื่องหมายคำถาม ใช้งานเหมือนเครื่องหมาย * เพียงแต่สามารถแทนตัวอักษรได้เพียงตัวเดียวเท่านั้น
[..]	ระบุกลุ่มตัวอักษรที่ต้องการภายในเครื่องหมายก้ามปู โดยแต่ละตัวอักษรจะต้องเว้นห่างกันด้วยเครื่องหมาย space ตัวอย่างเช่น [a b c d] จะหมายถึง เงื่อนไขที่ค่าของตัวแปรตรงกับตัวอักษร a ถึง d ค่าใดค่าหนึ่งก็ได้ อนึ่ง ผู้ใช้งานสามารถระบุค่าตัวอักษรที่เรียงลำดับติดกันได้ด้วยการใช้เครื่องหมาย - แทนที่จะต้องเขียนตัวอักษรทีละตัวในเครื่องหมายก้ามปู ตัวอย่างเช่น เขียน [a-d m x-z] ที่จะเป็น [a b c d m x y z] เป็นต้น

```
        then
          mv $tempfile bak.`basename $editfile`
          echo "$script: backup file created"
        else
          mv $tempfile editerr
          echo "$script: edit error--copy of \"plain
        "original file is in editerr" 1>&2
      fi
```

สคริปต์เริ่มต้นด้วยการตรวจสอบจำนวนอาร์กิวเมนต์ที่ผู้ใช้งานป้อนตามหลังคำสั่งโดยผ่านตัวแปรพิเศษ $# หากผู้ใช้งานมิได้ป้อนข้อความใด ๆ ต่อท้ายมาเลย คำสั่งควบคุม case ก็จะเลือกเงื่อนไขการทำงานไปที่ $#=0 ซึ่งเป็นการเรียกโปรแกรม vi ขึ้นมาเฉย ๆ โดยไม่ระบุชื่อไฟล์ที่ต้องการเปิดอ่านหรือสร้างขึ้นใหม่ ในกรณีที่ผู้ใช้งานเรียกใช้สคริปต์โดยป้อนอาร์กิวเมนต์ตามมาหนึ่งตัวซึ่งก็คือชื่อไฟล์ที่ต้องการเปิดอ่านหรือแก้ไขนั่นเอง คำสั่ง case ก็จะเลือกกระบวนการทำงานตามทางเลือกที่สอง โดยสคริปต์จะทำการตรวจสอบก่อนว่าไฟล์ดังกล่าวมีอยู่จริงหรือไม่ พร้อมกับตรวจสอบต่อไปว่าผู้ใช้งานมีสิทธิในการเปิดอ่านและเขียนลงบนไฟล์ดังกล่าวหรือไม่ และในกรณีที่ผู้ใช้งานเรียกใช้สคริปต์ safedit โดยมีอาร์กิวเมนต์ต่อท้ายมามากกว่าหนึ่งตัว คำสั่ง case ก็จะดำเนินการตามทางเลือกที่สามซึ่งเป็นการแสดงคำแนะนำในการใช้คำสั่งให้กับผู้ใช้งาน นอกจากการใช้ความสามารถของคำสั่ง case แล้ว สคริปต์ดังกล่าวยังแสดงให้เห็นถึงแนวทางปฏิบัติทั่วไปที่ผู้เขียนโปรแกรมเชลล์สคริปต์พึงกระทำในการเขียนโปรแกรมใช้งานจริงอยู่ 4 ประการด้วยกัน

ประการแรกก็คือการระบุตัวแปร PATH เพื่อกำหนดให้ทำการค้นหาภายในไดเรคทอรี /usr/ucb/bin และ /usr/bin ทั้งนี้เพื่อเป็นการเพิ่มความมั่นใจว่าสคริปต์จะสามารถทำงานได้ถูกต้องสมบูรณ์ การกำหนดตัวแปร PATH ไว้ภายในสคริปต์ก็เพื่อป้องกันความผิดพลาดในการที่ผู้ใช้งานอาจทำการกำหนดค่าตัวแปร PATH ให้เป็นไปตามที่ตนต้องการก่อนหน้าที่จะเรียกใช้สคริปต์ดังกล่าว หากไม่มีการกำหนด PATH ให้ถูกต้องภายในตัวสคริปต์เองแล้วก็อาจจะส่งผลให้การทำงานของสคริปต์เกิดการผิดพลาดไปได้

ประการที่สอง มีการสร้างตัวแปร script พร้อมกับกำหนดค่าให้เป็น "basename $0" เพื่อนำไปใช้อ้างอิงในส่วนต่าง ๆ ของเชลล์สคริปต์ คำสั่ง basename จะทำการส่งชื่อไฟล์อย่างง่ายของเชลล์สคริปต์ (ในที่นี้คือ safedit) เพื่อใช้ในการอ้างอิง ทั้งนี้เพราะในการใช้งานจริงผู้ใช้งานอาจทำการเรียกใช้สคริปต์ดังกล่าวในหลาย ๆ รูปแบบด้วยกัน ตัวอย่างเช่น

```
$ /home/pairoj/bin/safedit report
$ ./safedit report
$ safedit report
```

จะเห็นได้ว่าในแต่ละกรณีนี้ คำสั่ง safedit ก็ล้วนแต่ถูกเก็บไว้ในไดเรคทอรีเดียวกันคือ /home/pairoj/bin แต่วิธีเรียกจะแตกต่างกัน การระบุชื่อไฟล์อย่างง่ายจะทำให้ผู้ใช้งานไม่สับสนและเป็นการสร้างมาตรฐานในการอ้างอิงเดียวกัน กล่าวคือ ไม่ว่าผู้ใช้งานจะทำการป้อนคำสั่งเรียกใช้งานอย่างใดก็ตาม ต่างก็จะเห็นการแจ้งผลกลับในลักษณะเดียวกัน ตัวอย่างเช่น "safedit: backup file created" แทนที่จะเป็น "/home/pairoj/bin/safedit: backup file created" หรืออื่น ๆ เป็นต้น คำถามที่ตามมาก็คือทำไมไม่แทนที่ $script ด้วยคำว่า safedit ไปเลย ผู้เขียนขอตอบว่าในกรณีที่เราต้องการเปลี่ยนชื่อของสคริปต์ก็สามารถทำได้โดยการใช้คำสั่ง mv หรือ cp เพื่อเปลี่ยนชื่อสคริปต์ไปได้โดยที่ไม่ต้องมาพะวงถึงชื่อที่ระบุภายในสคริปต์ กล่าวคือ ไม่ว่าชื่อของสคริปต์จะเปลี่ยนเป็นอะไร การแสดงผลที่เกี่ยวโยงกับชื่อสคริปต์ก็จะเปลี่ยนชื่อไปโดยอัตโนมัตินั่นเอง

UNIX

ประการที่สาม ในกรณีที่ผู้ใช้งานทำการป้อนชื่อไฟล์ที่ต้องการแก้ไข และสคริปต์พบว่าผู้ใช้งานมีสิทธิในการแก้ไขไฟล์ดังกล่าวแล้ว ก่อนที่จะเรียกใช้โปรแกรม vi เพื่อเปิดไฟล์ สคริปต์จะทำการสำเนาไฟล์ดังกล่าวไปเก็บไว้เป็นไฟล์ชั่วคราว การตั้งชื่อไฟล์จะใช้ตัวแปรพิเศษ $$ ซึ่งจะให้เลขหมายประจำโปรเซส (PID) ในขณะนั้นกลับมา รูปแบบการตั้งชื่อไฟล์จะเป็น /tmp/$$.$script ซึ่งผู้ใช้งานอาจรู้สึกไม่คุ้นเคยกับการที่นำชื่อไฟล์ไว้ข้างหลังเครื่องหมาย "." ข้อดีของการตั้งชื่อในลักษณะดังกล่าวสืบเนื่องมาจากการที่ระบบปฏิบัติการรองรับการใช้งานจากผู้ใช้งานหลายคน บางครั้งอาจมีผู้ใช้งานสองคนดำเนินกระบวนการบางอย่างที่ก่อให้เกิดการสร้างไฟล์ชั่วคราว (temporary file) ที่มีชื่อเหมือนกันลงในไดเรคทอรี /tmp ซึ่งจะก่อให้เกิดความเสียหายกับไฟล์ชั่วคราวของผู้ที่ถูกเขียนทับได้ จึงนิยมเติมเลขหมาย PID ลงเป็นส่วนหนึ่งของชื่อไฟล์ด้วย ทั้งนี้เนื่องจากเลขหมายประจำโปรเซสย่อมไม่มีทางซ้ำกัน สำหรับสาเหตุของการนำเลขหมาย PID มาไว้ข้างหน้าก็เนื่องมาจากระบบปฏิบัติการ UNIX บางรุ่นรองรับการตั้งชื่อไฟล์ได้ยาวไม่เกิน 14 ตัวอักษร หากนำเลขหมาย PID ไปต่อท้ายชื่อไฟล์ ในกรณีที่ไฟล์มีชื่อยาวใกล้เคียงกับ 14 ตัวอักษรก็อาจทำให้เกิดปัญหาการตั้งชื่อซ้ำกันขึ้นได้อีก เนื่องจากเลขหมาย PID จะถูกตัดออกไป เพื่อเป็นการหลีกเลี่ยงปัญหาดังกล่าวจึงนิยมตั้งชื่อไฟล์ชั่วคราวในลักษณะนี้

ประการที่สี่ ให้สังเกตการกำหนดเงื่อนไขให้กับคำสั่งควบคุม if... then... else ชุดสุดท้าย จะเห็นว่าเงื่อนไขที่ใช้คือ vi $editfile ซึ่งหากการแก้ไขไฟล์ได้ผ่านไปอย่างสมบูรณ์และผู้ใช้งานได้ออกจากโปรแกรม vi แล้ว ค่าย้อนกลับที่คำสั่ง if... then... else จะได้รับจากโปรแกรม vi โดยทั่วไปคือ 0 ซึ่งจะทำให้สคริปต์ดำเนินการต่อไปตามกระบวนการหลังสเตทเมนต์ then แต่หากในกรณีที่การออกจากโปรแกรม vi เป็นไปอย่างผิดปกติ เช่น อาจจะเกิดจากการที่ผู้ใช้งานยกเลิกคำสั่ง vi โดยการส่งคำสั่ง kill มาจากอุปกรณ์เทอร์มินัลเครื่องอื่น เช่นนี้จะทำให้ค่าย้อนกลับที่คำสั่ง if... then... else ได้รับจะเป็น 1 ซึ่งทำให้สคริปต์ดำเนินการตามเงื่อนไขระหว่างสเตทเมนต์ else และ fi แทน

สำหรับบทความในตอนต่อไปซึ่งจะเป็นตอนจบของการเขียนโปรแกรมเชลล์สคริปต์บนเชลล์แบบบอร์น ผู้เขียนจะกล่าวถึงเรื่องของการใช้คำสั่ง trap และ exec ตามด้วยการเรื่องของชุดฟังก์ชันมาตรฐานที่มีมาพร้อมกับเชลล์ชนิดนี้ แล้วพบกันใหม่ครับ...